2D Convolution

Introduction

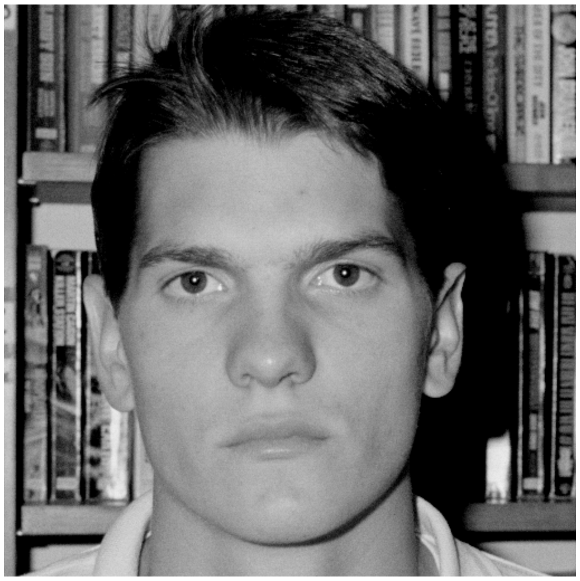
This project has three objectives

1. To develop experience performing digital signal processing
2. To solidify understanding of two-dimensional convolution
3. To observe the effects of three different kinds of image filters

The filters that were used were

1. A simple low-pass filter
2. A modified Sobel filter
3. A matched correlation filter

The image to the right was used in all filters, and processing was done in c++

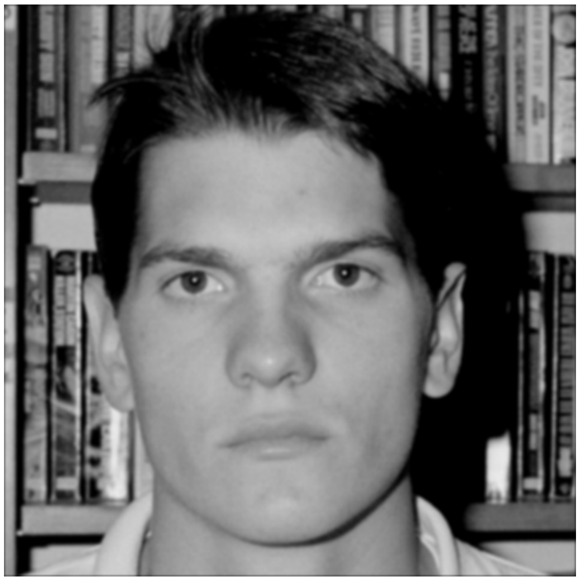


The Low Pass Filter

The low-pass filter below was used to produce the image to the right

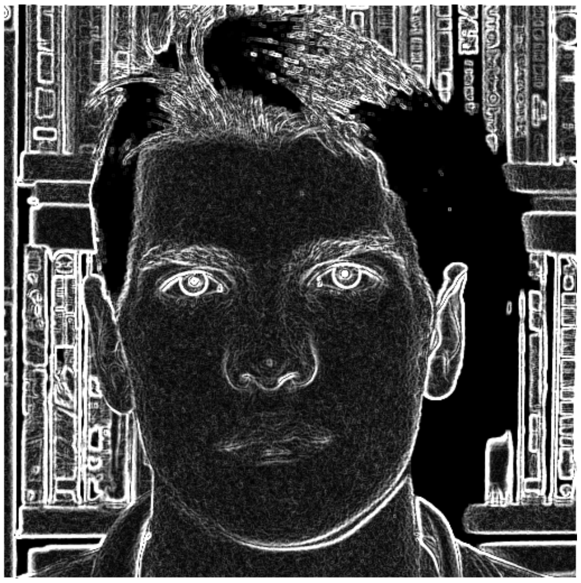
$$\mathbf{H}_1 = \frac{1}{81}\begin{bmatrix} 1 & 2 & 3 & 2 & 1 \\ 2 & 4 & 6 & 4 & 2 \\ 3 & 6 & 9 & 6 & 3 \\ 2 & 4 & 6 & 4 & 2 \\ 1 & 2 & 3 & 2 & 1 \end{bmatrix}$$

As you can see the image to the right is a little more “fuzzy” or “blurry” than the picture above. The sharper, high frequency content has been filtered out.



The Sobel Filter

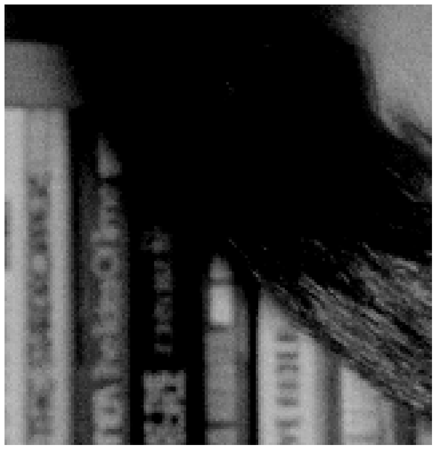
A Sobel filter is comprised of two filters, one that detects edges horizontally and one that detects them vertically. The filters are applied separately and the results superimposed on each other. Traditional Sobel filters use a square root of sum of squares for each component. For this project the absolute value of each component is added together. The filters which were used to produce the image on the right are seen below.

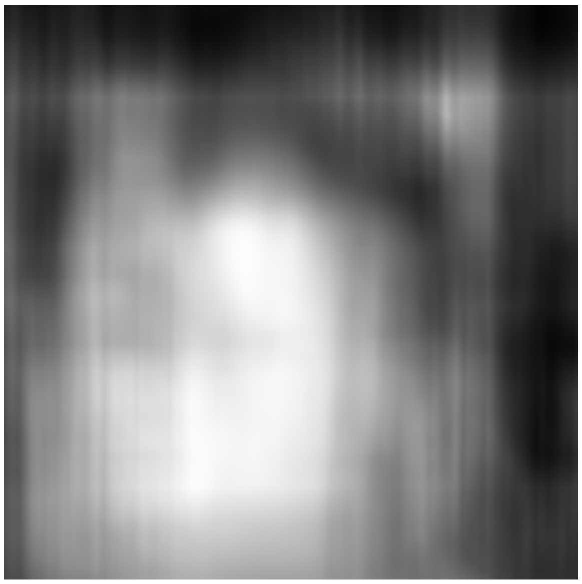


$$\mathbf{S}_1 = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \qquad \mathbf{S}_2 = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

The Matched Correlation Filter

Correlation is the same operation as convolution, only the filter isn't inverted. Thus using an inverted filter with convolution yields the same operation. The filter on the left was used to produce the image on the right. As you can see it is a rotated section of the top right corner of the original image, resulting in a "white spot" in the top right of the filtered image. Apparently large sections of the middle of the image also correlate highly with the filter.





Conclusion

Images can be filtered using 2D convolution. There are many types of filters with a wide variety of applications. Convolution is at the center of digital signal and image processing.

Appendix: C++ Code

image.hpp

```
#pragma once
#include <functional>
#include <string>

typedef unsigned char byte;
#define ERROR_NONE 0

template <typename T>
class Image {
private:
	inline void clean() {
		delete[] image;
		image = nullptr;
		width = height = length = 0;
	}
	inline void copy(const Image<T>& rhs) {
		width = rhs.width;
		height = rhs.height;
		length = width * height;
		image = new T[length];
		memcpy(image, rhs.image, sizeof(T) * length);
	}

protected:
	int width, height, length;
	T* image;

	Image() {
		width = 0;
		height = 0;
		length = 0;
		image = nullptr;
	}

public:
	friend int OpenPGM(std::string, Image<byte>**);
	friend int SavePGM(std::string, Image<byte>*);

	typedef std::function<void(int, int, T)> accessor;
	typedef std::function<T(int, int, T)> mutator;
	typedef std::function<T(int, int)> setter;

	inline void each(accessor f) const {
		int i = 0;
		for (int n = 0; n < height; ++n) {
			for (int m = 0; m < width; ++m) {
				f(m, n, image[i++]);
			}
		}
	}
	inline void each(mutator f) {
		int i = 0;
		for (int n = 0; n < height; ++n) {
			for (int m = 0; m < width; ++m) {
				image[i] = f(m, n, image[i]);
				++i;
			}
		}
	}
	Image(int w, int h, setter f) {
		width = w;
		height = h;
		length = width * height;
		image = new T[length];
		int i = 0;
		for (int n = 0; n < height; ++n) {
			for (int m = 0; m < width; ++m) {
				image[i++] = f(m, n);
			}
		}
	}

	Image(int w, int h) {
		width = w;
		height = h;
		length = width * height;
		image = new T[length]();
	}
```

```
		Image(int w, int h, T v) {
			width = w;
			height = h;
			length = width * height;
			image = new T[length];
			int i = 0;
			for (int n = 0; n < height; ++n) {
				for (int m = 0; m < width; ++m) {
					image[i++] = v;
				}
			}
		}
		Image(int w, int h, T* img) {
			width = w;
			height = h;
			length = width * height;
			image = new T[length];
			memcpy(image, img, sizeof(T) * length);
		}
		Image<T>& operator=(const Image<T>& rhs) { clean(); copy(rhs); return *this; }
		Image(const Image<T>& rhs) { copy(rhs); }
		~Image() { clean(); }

		inline const int M() const { return width; }
		inline const int N() const { return height; }

		inline const int ConvTailM() const { return width / 2; }
		inline const int ConvTailN() const { return height / 2; }

		inline T Get(int m, int n) const {
			if (m < 0 || n < 0 || m >= width || n >= height || image == nullptr) { return 0; }
			return image[width*n + m];
		}
		inline void Set(int m, int n, T val) {
			if (m < 0 || n < 0 || m >= width || n >= height || image == nullptr) { return; }
			image[width*n + m] = val;
		}
};

#define ERROR_PGM_FILE          (1 << 0)
#define ERROR_PGM_HEADER        (1 << 1)
#define ERROR_PGM_MAXSIZE       (1 << 2)

int OpenPGM(std::string, Image<byte>**);
int SavePGM(std::string, Image<byte>*);
```

image.cpp

```
#include "image.hpp"
#include <fstream>

enum ParsePGM {
	Width,
	Height,
	MaxVal,
	Data,
};

int OpenPGM(std::string file, Image<byte>** out) {
	std::fstream fin(file, std::ios::binary | std::ios::in);
	if (!fin) {
		return ERROR_PGM_FILE;
	}

	char in;
	bool error = false;

	// Validate the magic numbers
	fin.read(&in, 1);
	if (in == 'P') {
		fin.read(&in, 1);
		if (in == '5') {
			fin.read(&in, 1);
			if (in != '\n') {
				error = true;
			}
		}
		else {
			error = true;
		}
	} else {
		error = true;
	}

	if (error) {
		fin.close();
		return ERROR_PGM_HEADER;
	}

	// Do the actual parsing
	Image<byte>* iin = new Image<byte>();

	ParsePGM state = ParsePGM::Width;
	bool comment = false;
	bool newline = true;
	int maxsize = 0;

	while (true) {
		fin.read(&in, 1);

		if (newline) {
			newline = false;
			if (in == '#') {
				comment = true;
				continue;
			}
		}

		if (comment) {
			if (in == '\n') {
				comment = false;
				newline = true;
			}
			continue;
		}

		switch (state) {
			case ParsePGM::Width:
				if (in == ' ' || in == '\n' || in == '\t' || in == 0) {
					state = ParsePGM::Height;
					break;
				}
				if (in < '0' || in > '9') {
					delete iin;
					fin.close();
					return ERROR_PGM_HEADER;
				}
				iin->width *= 10;
				iin->width += in - '0';
				break;

			case ParsePGM::Height:
				if (in == ' ' || in == '\n' || in == '\t' || in == 0) {
					state = ParsePGM::MaxVal;
					break;
				}
```

```
                                        if (in < '0' || in > '9') {
                                                delete iin;
                                                fin.close();
                                                return ERROR_PGM_HEADER;
                                        }
                                        iin->height *= 10;
                                        iin->height += in - '0';
                                        break;

                                case ParsePGM::MaxVal:
                                        if (in == ' ' || in == '\n' || in == '\t' || in == 0) {
                                                state = ParsePGM::Data;
                                                if (maxsize != 255) {
                                                        delete iin;
                                                        fin.close();
                                                        return ERROR_PGM_HEADER | ERROR_PGM_MAXSIZE;
                                                }

                                                iin->length = iin->width * iin->height;
                                                iin->image = new byte[iin->length];
                                                break;
                                        }
                                        if (in < '0' || in > '9') {
                                                delete iin;
                                                fin.close();
                                                return ERROR_PGM_HEADER;
                                        }
                                        maxsize *= 10;
                                        maxsize += in - '0';
                                        break;
                        }

                        if (state == ParsePGM::Data) { break; }
                }

                fin.read((char*)iin->image, iin->length);
                fin.close();

                *out = iin;
                return 0;
}

int SavePGM(std::string file, Image<byte>* in) {
                std::fstream fout(file, std::ios::binary | std::ios::out);
                fout << "P5\n" << in->width << ' ' << in->height << " 255\n";
                fout.write((char*)in->image, in->length);
                fout.close();
                return 0;
}
```

main.cpp

```cpp
#include <iostream>
#include <thread>
#include "image.hpp"

float H1[] = {
	1 / 81.0f, 2 / 81.0f, 3 / 81.0f, 2 / 81.0f, 1 / 81.0f,
	2 / 81.0f, 4 / 81.0f, 6 / 81.0f, 4 / 81.0f, 2 / 81.0f,
	3 / 81.0f, 6 / 81.0f, 9 / 81.0f, 6 / 81.0f, 3 / 81.0f,
	2 / 81.0f, 4 / 81.0f, 6 / 81.0f, 4 / 81.0f, 2 / 81.0f,
	1 / 81.0f, 2 / 81.0f, 3 / 81.0f, 2 / 81.0f, 1 / 81.0f,
};

float S1[] = {
	1, 0, -1,
	2, 0, -2,
	1, 0, -1,
};

float S2[] = {
	-1, -2, -1,
	 0,  0,  0,
	 1,  2,  1,
};

// Naive Approach
template<typename T1, typename T2>
Image<float>* Convolve2D(Image<T1>* image, Image<T2>* filter) {
	int M = image->M() + filter->M() - 1;
	int N = image->N() + filter->N() - 1;
	return new Image<float>(M, N, [image, filter](int m, int n) -> float {
		float sum = 0;
		filter->each([image, m, n, &sum](int l, int k, T2 v) -> void {
			sum += (float)v * image->Get(m - l, n - k);
		});
		return sum;
	});
}

// Optimization 1 - Remove std::function from 4-layer loops
template<typename T1, typename T2>
Image<float>* O1Convolve2D(Image<T1>* image, Image<T2>* filter) {
	int MI = image->M();
	int NI = image->N();

	int MF = filter->M();
	int NF = filter->N();

	int M = MI + MF - 1;
	int N = NI + NF - 1;

	Image<float>* output = new Image<float>(M, N);
	for (int n = 0; n < N; ++n) {
		for (int m = 0; m < M; ++m) {
			float sum = 0;
			for (int k = 0; k <= n && k < NF; ++k) {
				for (int l = 0; l <= m && l < MF; ++l) {
					sum += (float)filter->Get(l, k) * image->Get(m - l, n - k);
				}
			}
			output->Set(m, n, sum);
		}
	}
	return output;
}

// Struct helper for Multithreading in Optimization 2
template<typename T1, typename T2>
class Plan {
public:
	Image<T1>* I;
	Image<T2>* F;
	int MI, NI;
	int MF, NF;
	int M, N;
	Image<float>* out;

	Plan(Image<T1>* image, Image<T2>* filter) {
		I = image; F = filter;

		MI = I->M();
		NI = I->N();

		MF = F->M();
		NF = F->N();

		M = MI + MF - 1;
		N = NI + NF - 1;
```

```
			out = new Image<float>(M, N);
		}
};

// Thread routine for Multithreading in Opimization 2 (compare loop with O1Convolve2D)
template<typename T1, typename T2>
void FastConvolve2DThread(Plan<T1, T2>* plan, int n0, int n1) {
		if (n1 > plan->N) { n1 = plan->N; }
		for (int n = n0; n < n1; ++n) {
			for (int m = 0; m < plan->M; ++m) {
				float sum = 0;
				for (int k = 0; k <= n && k < plan->NF; ++k) {
					for (int l = 0; l <= m && l < plan->MF; ++l) {
						sum += (float)plan->F->Get(l, k) * plan->I->Get(m - l, n - k);
					}
				}
				plan->out->Set(m, n, sum);
			}
		}
}

// The number of threads to use in optimization 2 (divided roughly equally, the last one may be slightly less workload)
#define POOL_SIZE 10

// Optimization 2 - Multithreaded computation
template<typename T1, typename T2>
Image<float>* FastConvolve2D(Image<T1>* image, Image<T2>* filter) {
		Plan<T1, T2> plan(image, filter);

		std::thread* pool[POOL_SIZE];
		int dn = plan.N / 10 + 1;

		for (int i = 0; i < POOL_SIZE; ++i) {
			int n0 = i * dn;
			int n1 = n0 + dn;
			pool[i] = new std::thread(FastConvolve2DThread<T1, T2>, &plan, n0, n1);
		}

		for (int i = 0; i < POOL_SIZE; ++i) {
			pool[i]->join();
			delete pool[i];
			pool[i] = nullptr;
		}

		return plan.out;
}

int main() {
		int err;

		Image<float>* H1Filter = new Image<float>(5, 5, H1);
		Image<float>* S1Filter = new Image<float>(3, 3, S1);
		Image<float>* S2Filter = new Image<float>(3, 3, S2);

		Image<byte>* image;
		Image<byte>* filter;

		err = OpenPGM("image.pgm", &image);
		if (err != ERROR_NONE) {
			std::cout << "Unable to parse image.pgm! Error Code: " << err << std::endl;
			exit(EXIT_FAILURE);
		}

		err = OpenPGM("filter_final.pgm", &filter);
		if (err != ERROR_NONE) {
			std::cout << "Unable to parse filter_final.pgm! Error Code: " << err << std::endl;
			exit(EXIT_FAILURE);
		}

		// Testing to verify that we save the same images we read
		//err = SavePGM("test_image.pgm", image);
		//if (err != ERROR_NONE) {
		//		std::cout << "Unable to save test_image.pgm! Error Code: " << err << std::endl;
		//		exit(EXIT_FAILURE);
		//}
		//err = SavePGM("test_filter.pgm", filter);
		//if (err != ERROR_NONE) {
		//		std::cout << "Unable to save test_filter.pgm! Error Code: " << err << std::endl;
		//		exit(EXIT_FAILURE);
		//}

		// Problem 2
		Image<float>* H1F = FastConvolve2D(image, H1Filter);
		Image<byte>* P2 = new Image<byte>(image->M(), image->N(), [H1F, H1Filter](int m, int n) -> byte {
			float h1 = H1F->Get(m + H1Filter->ConvTailM(), n + H1Filter->ConvTailN()); // Trim convolution tails by shifting
			if (h1 > 255) { h1 = 255; }
			if (h1 < 0) { h1 = 0; }
			return (byte)h1;
		});
		err = SavePGM("P2.pgm", P2);
		if (err != ERROR_NONE) {
			std::cout << "Unable to save P2.pgm! Error Code: " << err << std::endl;
			exit(EXIT_FAILURE);
		}

		// Problem 3
		Image<float>* G1 = FastConvolve2D(image, S1Filter);
		Image<float>* G2 = FastConvolve2D(image, S2Filter);
		Image<byte>* P3 = new Image<byte>(image->M(), image->N(), [G1, G2, S1Filter, S2Filter](int m, int n) -> byte {
			float g1 = G1->Get(m + S1Filter->ConvTailM(), n + S1Filter->ConvTailN()); // Trim convolution tails by shifting
			float g2 = G2->Get(m + S2Filter->ConvTailM(), n + S2Filter->ConvTailN()); // Trim convolution tails by shifting
			g1 = g1 < 0 ? g1 * -1 : g1;
			g2 = g2 < 0 ? g2 * -1 : g2;
			float sum = g1 + g2;
			if (sum > 255) { sum = 255; }
			return (byte)sum;
		});
		err = SavePGM("P3.pgm", P3);
		if (err != ERROR_NONE) {
			std::cout << "Unable to save P3.pgm! Error Code: " << err << std::endl;
			exit(EXIT_FAILURE);
		}

		// Problem 4

		// Before you filter, subtract a scalar value from each value in the filter
		// The scalar is the minimim value in the filter
		// It so happens this value is ... 0?
		//byte minScalar = 255;
		//filter->each([&minScalar](int m, int n, byte v) -> void {
		//		if (v < minScalar) { minScalar = v; }
		//});
		//filter->each([minScalar](int m, int n, byte v) -> byte {
		//		return v - minScalar;
		//});

		Image<float>* F1 = FastConvolve2D(image, filter);

		// Scale the filtered image so the maximum value in the image is 255
		// and all negative values after scaling are set to zero (done in ctor() of P4)
		//float minF1;
		float maxF1 = F1->Get(0, 0);
		F1->each([&maxF1](int m, int n, float v) -> void {
			//if (v < minF1) { minF1 = v; }
			if (v > maxF1) { maxF1 = v; }
		});
		F1->each([maxF1](int m, int n, float v) -> float {
			float scaled = v * 255 / maxF1;
			return scaled;
		});

		Image<byte>* P4 = new Image<byte>(image->M(), image->N(), [F1, filter](int m, int n) -> byte {
			float f1 = F1->Get(m + filter->ConvTailM(), n + filter->ConvTailN()); // Trim convolution tails by shifting
			if (f1 > 255) { f1 = 255; }
			if (f1 < 0) { f1 = 0; }
			return (byte)f1;
		});
		err = SavePGM("P4.pgm", P4);
		if (err != ERROR_NONE) {
			std::cout << "Unable to save P4.pgm! Error Code: " << err << std::endl;
			exit(EXIT_FAILURE);
		}

		delete image;
		delete filter;
		return 0;
}
```